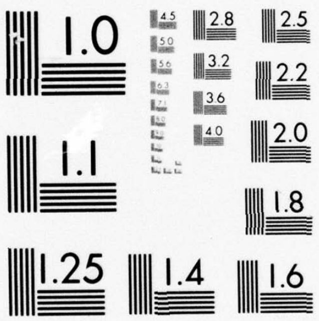
AD-A039 566 YALE UNIV NEW HAVEN CONN DEPT OF COMPUTER SCIENCE F/G 12/1
SYNCHRONIZATION AND COMPUTING CAPABILITIES OF LINEAR ASYNCHRONO--ETC(U)
OCT 75 R J LIPTON, R E MILLER, L SNYDER N00014-75-C-0752
UNCLASSIFIED RC-5857 NL

1 OF 1
AD
A039566

END
DATE
FILMED
6-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0

AD A039566

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) RC-5857 ✓	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) Synchronization and Computing Capabilities of Linear Asynchronous Structures.		5. TYPE OF REPORT & PERIOD COVERED (9) Technical rept.
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Richard J./Lipton, Raymond E./Miller, Lawrence/Snyder		8. CONTRACT OR GRANT NUMBER(s) (15) N00014-75-C-0752, NSF-DCR-74-12870
9. PERFORMING ORGANIZATION NAME AND ADDRESS Yale University, Department of Computer Science, 10 Hillhouse Ave, New Haven, CT 06520		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (12) 43p.
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, Information Systems Program, Arlington, Virginia 22217		12. REPORT DATE (11) OCT 1975
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

(12)

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

parallel systems
linear asynchronous structures
firing squad synchronization problem
Church Rosser Theorem
delay
cellular arrays

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A model is defined in which questions concerning delay bounded asynchronous parallel systems may be investigated. It is shoen that synchronization problems, similar to the "firing squad synchronization problem," cannot be solved by delay bounded asynchronous systems. Three conditions called persistence, determinacy, and single change are introduced. These conditions are shown to be sufficient to guarantee that a synchronous execution policy can be relaxed to an asynchronous execution policy with no change to the result of the computation. This is a Church-Rosser type theorem, but in addition, the

→ next page

407 051 ✓

#20

asynchronous execution time is shown to be only (D+1) times the synchronous execution time, where D is the delay bound. Finally, a wide class of recognition problems is identified which can be solved by linear asynchronous structures.

RC 5857
(#25330)
2/11/76
Computer
Science

40 pages

SYNCHRONIZATION AND COMPUTING CAPABILITIES OF LINEAR ASYNCHRONOUS STRUCTURES[†]

R. J. Lipton[*]
Department of Computer Science
Yale University
10 Hillhouse Avenue
New Haven, Connecticut 06520

R. E. Miller
Mathematical Sciences Department
IBM Thomas J. Watson Research Center
P. O. Box 218
Yorktown Heights, New York 10598

L. Snyder[**]
Department of Computer Science
Yale University
10 Hillhouse Avenue
New Haven, Connecticut 06520

ABSTRACT: A model is defined in which questions concerning delay bounded asynchronous parallel systems may be investigated. It is shown that synchronization problems, similar to the "firing squad synchronization problem," cannot be solved by delay bounded asynchronous systems. Three conditions called persistence, determinacy, and single change are introduced. These conditions are shown to be sufficient to guarantee that a synchronous execution policy can be relaxed to an asynchronous execution policy with no change to the result of the computation. This is a Church-Rosser type theorem, but in addition, the asynchronous execution time is shown to be only (D+1) times the synchronous execution time, where D is the delay bound. Finally, a wide class of recognition problems is identified which can be solved by linear asynchronous structures.

[†] A preliminary version of this paper was presented at the IEEE Symposium on Foundations of Computer Science, Berkeley, California, October 1975.

[*] [**] Supported in part by Office of Naval Research under grant N00014-75-C-0752.

[*] Also, supported in part under NSF grant DCR-74-12870, and part of this work was done while visiting IBM Research Center.

Copies may be requested from:
IBM Thomas J. Watson Research Center
Post Office Box 218
Yorktown Heights, New York 10598

1. Introduction

Computational systems, whether they be hardware or software, are usually envisioned as an interconnection of a number of separate and distinct processes. Each of the processes is assumed to perform a particular task, obtaining inputs from other processes in the system and providing results to other processes in the system. The function of the whole system is accomplished through the combined effort of the distinct processes acting in concert. A specification of the overall control of when processes are to act and communicate with each other is usually required to insure proper operation of the system. In programs this is usually done by specifying the "flow of control" of the program, whereas in hardware this is usually done by having a centralized control unit which emits control signals to the processes. As is well known, efficiencies can often be realized by having several processes act simultaneously, or in parallel, rather than having a single sequence of process actions. Such parallel computation, however, is often quite complex to control, especially when the time of process performance is variable.

In this paper we study the intercommunication problems for systems of interconnected processes, acting in parallel, where the time required for a process to act is not known exactly. As a simplifying assumption we restrict our attention here to linear interconnection of processes. The results we obtain are then directly applicable to such "linear structures." Also this provides some information about systems having more complex interconnection since in any such system there are linear chains of interconnected processes. The linearity assumption allows us to draw on, and compare our results with, the extensive work done on synchronous linear structures. For example, the "Firing Squad Synchronization Problem" [7] is one of the earliest questions studied in this context. Since that time, a large quantity of literature has appeared on cellular machines, iterative arrays, parallel grammars, L-systems, etc. [2,4,9,10]. All of these studies assume synchronous computation. That is, at each discrete moment in time, if a machine can perform some transition (or, in the grammatical case, if a production applies) then that transition must be performed. The consequences of relaxing the synchronous requirement to asynchronous operation are: first, that some tasks which can be done synchronously cannot even be approximated asynchronously, and second, for those synchronous computations that can be realized asynchronously, the previously used techniques fail to apply and a new set of techniques must be developed. The asynchronous assumption is a useful one to make since often processes have execution times which depend upon the data. We do assume that the times are known to be within some upper and lower bounds, although they may vary with time within these bounds. Examples where such situations arise include both cellular arrays of devices, where each device runs at some nonzero rate, and operating systems, where each process is given a nonzero, but somewhat variable, amount of time to act.

Our model, to be formally defined in the next section, hypothesizes a system of n identically structured finite state machines organized as a linear array. Each machine is allowed to communicate with other machines in its own neighborhood (not necessarily just with its adjacent neighbors). The time is measured in a relative fashion, with one step elapsing whenever some machine(s) change state. A given machine is said to become active when it is first capable of a transition. (Identity transitions are not allowed, so a device may not be capable of another state change immediately after a transition has taken place.) Once active, the machine can perform the state change at any step. However, no machine can remain active, without changing state, for more than D steps. The delay, D, is a nonnegative integral value which gives the number of steps any processor is allowed to remain idle prior to completing a computational step. Hence, when $D = 0$, no idle steps are allowed, each processor completes execution at each step and, therefore, the system is synchronous. When $D > 0$, the system is asynchronous and the processors operate at a worst case rate of once every $D + 1$ steps.

Clearly, because the rate of execution is a parameter, the model to be described will be equally capable of characterizing synchronous, as well as asynchronous, computation. Indeed, by varying D, a single system can be executed using either policy. This facilitates our study of the relationships between synchronous and asynchronous parallel computation.

Several comments are in order. First, note that no assumption is made as to whether or not the relative time steps are of equal length. Furthermore, no assumption is made about how long it takes for a given device to change state, except that it is bounded. Consequently, we are allowing the execution time of a given device to change for any reason whatsoever. The same transition can even take a different number of steps for different devices or for the same device at different points in the computation. All that is required is that it be bounded by $D + 1$ (D is fixed for any given computation). This point of view is motivated by an interest in modelling parallel circuits as well as operating systems. In the former case, the performance of the device may be influenced by physical characteristics of the components. In the latter case, a process may be influenced by competition with other processes for resources, or influenced by I/O or some other exogenous variables. In any case, if the delay D cannot be chosen precisely for a given system, then it may be considered to be a limit beyond which the failure of a processor to execute is interpreted as a failure of the entire system.

A second observation is that the assumption of "identically structured" processes is not overly restrictive. The assumption should probably be stated as "identically structured with respect to the interaction among processes." Hence, the interaction of multiple instances of processes which communicate in the same manner is being studied. Any computation not relevant to this communication is allowed; since it doesn't influence the overall synchronization behavior, however, it can be ignored.

Finally, a word of warning is in order about the role of D. D, as it is used in the sequel, is the delay, or the number of idle steps allowed before a device must execute. Consequently, the "firing frequency" for processors which are always active will, in the worst case, be once every $D + 1$ steps. Thus, for the synchronous case, $D = 0$, the devices must fire at each step and, therefore, no idle steps are allowed for active processors.

The main question addressed in this paper is:

> How do linear arrays of machines operating synchronously compare with linear arrays of machines operating asynchronously in terms of computational and synchronization characteristics?

First of all we note that observed globally, a synchronous array has precisely one execution sequence (assuming, as we do, that the machines are deterministic). By contrast, an asynchronous array defines a set of computations corresponding to the differing execution rates of the individual machines. Obviously, one of these computations is a "synchronous" computation (in the sense that each machine executes without any delay). Thus, if we consider an asynchronous computation to be well behaved if the computed result is independent of the individual execution rates, then clearly, anything that can be computed asynchronously can be computed synchronously. Our main question thus reduces to: are asynchronous arrays weaker than synchronous arrays? The answer depends upon whether we speak of synchronization ability or computational ability.

It is known that cellular arrays can solve a synchronization problem known as the "firing squad synchronization problem," [7]. It would be foolish to expect an asynchronous linear array to solve this problem (for $D > 0$) since the soldiers may or may not choose to "fire" at the appointed moment. But suppose that we required all soldiers to "fire" within an interval of size D. It will be shown that this simpler problem cannot be solved! Indeed, a stronger result will be shown. Hence, with respect to synchronizing qualities, the asynchronous linear arrays are weaker than their synchronous counterparts.

By contrast, it will be shown that for language recognition problems, asynchronous arrays are no weaker than the synchronous linear arrays. This is unexpected since in the synchronous arrays the techniques used to solve the firing squad problem are central to the solution of recognition problems.

In [10] the recognition questions were analyzed in terms of the time required by the linear array. Hence, it is not only of interest whether a particular set can be recognized, but the time required in comparison to the synchronous case is also relevant. We show that it takes at most 3(D+1) times longer.

This last result uses another of our main theorems. Namely, we identify three properties of linear asynchronous systems -- determinacy, persistence, and single change -- and show that these are sufficient to guarantee that any system with these properties operates asynchronously at most (D+1) times slower than it does synchronously, for all D. With this result we obtain an *effective strategy* for solving a problem with asynchronous systems: First find a synchronous system for the task. Establish determinacy, persistence, and single change, and then invoke the above theorem. The validity and performance are thus established.

The format of the remainder of the paper is as follows. Section 2 gives initial definitions and illustrative examples. Section 3 shows the impossibility of a linear asynchronous system solving the firing squad synchronization problem. Section 4 proves the CR Theorem* on the synchronous to asynchronous relationship. Section 5 establishes the equivalence between synchronous and asynchronous recognition, and Section 6 poses some open problems.

* The name is motivated by the fact that this theorem has a flavor similar to the Church-Rosser Theorem of Lambda Calculus. [1,3].

2. Basic Definitions and Examples

In this section we introduce the basic model, present examples and provide further motivation.

Although we have purposely chosen a model that is closely related to the iterative arrays and cellular automata models so as to provide convenient comparison, we have not used the finite state machine as a basic constituent of the model. Instead, we avoid the cumbersome details of these machines by basing the model on a rewriting system that we call an asynchronous grammar. Even so, we will continue to employ the machine metaphor in our informal discussions; first because it is a handy conceptual tool, and second because we believe the work includes application to asynchronous systems that are actually implemented as circuits.

Definition 2.1. An asynchronous grammar $G = <\Sigma, P>$ consists of a finite alphabet Σ and a finite set P of productions of the form $\alpha \rightarrow \beta$ where

(i) $\alpha, \beta \in \Sigma^*$

(ii) $|\alpha| = |\beta|$ [†]

(iii) $\alpha \neq \beta$.

Hence, an asynchronous grammar is a rewriting system with length preserving, non-identity productions over the alphabet. The state of a linear array of n machines is thus represented by a word $x_1x_2\cdots x_n \in \Sigma^n$. [††]

[†] $|\alpha|$ denotes the length of string α

[††] Subscripts are used to denote coordinate positions.

A rule $\alpha \rightarrow \beta$ models changes in the state of machines in terms of their own states and the states of neighboring machines. The $\alpha \neq \beta$ constraint is intuitively natural since in an asynchronous system the only detectable action is a change in a state. Remaining in a state is considered as no action.

Given a string $x_1 \cdots x_n$, suppose there is a production $\alpha \rightarrow \beta$ such that

$$\alpha = x_i\, x_{i+1} \cdots x_j .$$

Then $\alpha \rightarrow \beta$ can be applied to $x_1 \cdots x_n$, and if $\beta = y_i\, y_{i+1} \cdots y_j$ we obtain the string $x_1 \cdots x_{i-1}\, y_i\, y_{i+1} \cdots y_j\, x_{j+1} \cdots x_n$. Thus, successive modifications to some initial string in Σ^n will represent the successive state changes in the linear asynchronous structure.

A particularly simple case is that for an ordinary array of n machines where any machine's next state depends only upon its current state and the current states of its two neighbors. This situation is represented by a grammar in which $|\alpha| = 3$. The middle symbol representing the machine under consideration and the two end symbols represent the neighbors. Thus a production $\alpha \rightarrow \beta$ would take the form: $a_1a_2a_3 \rightarrow b_1b_2b_3$, where $a_1 = b_1$, $a_3 = b_3$ and $a_2 \neq b_2$ indicating the desired change.

A precise description of how changes can occur due to production application is:

Definition 2.2. Let $x = x_1 \cdots x_n \in \Sigma^n$ and $y = y_1 \cdots y_n \in \Sigma^n$. We write $x \vdash y$ provided $x \neq y$ and $x_i \neq y_i$ implies there exists a production $\alpha_1 \cdots \alpha_k \rightarrow \beta_1 \cdots \beta_k \in P$ such that

(i) $\exists j,\ 1 \le j \le n$ such that $x_{i-j+1} \cdots x_i \cdots x_{i-j+k}$

$= \alpha_1 \cdots \alpha_j \cdots \alpha_k$, and

(ii) $y_{i-j+s} = \beta_s$ whenever $\alpha_s \neq \beta_s$ for $s = 1,2,\cdots,k$.

Informally, $x_1 \cdots x_n \vdash y_1 \cdots y_n$ if wherever a change takes place $(x_i \neq y_i)$, then there is some production matching some context around x_i (requirement (i)) and that each change implied by the production $(\alpha_s \neq \beta_s)$ is reflected in the result $(\beta_s = y_{i-j+s})$. This definition is quite general, allowing overlapping application of productions.

Example 2.1. Let $G = \langle\{1,2,3\}, \{12 \to 13,\ 23 \to 21,\ 12 \to 33\}\rangle$. Then the following are allowed:

$123 \vdash 133$
$123 \vdash 121$
$123 \vdash 333$
$123 \vdash 331$

Note that consistent overlapping is allowed and that in the last two cases, it is ambiguous whether $12 \to 13$ is applied, since it is subsumed by $12 \to 33$.

Example 2.2: Let $G = \langle\{1,2\}, \{12 \to 21\}\rangle$ then

$112122 \vdash 121122$
$112122 \vdash 112212$
$112122 \vdash 121212$

Note that this grammar is non-overlapping. We shall have more to say about this grammar later.

Definition 2.3. Given $G = \langle\Sigma, P\rangle$ and $x_1 \cdots x_n \in \Sigma^n$ coordinate i is active in $x_1 \cdots x_n$ provided there exists a $y_1 \cdots y_n \in \Sigma^n$ such that $x_1 \cdots x_n \vdash y_1 \cdots y_n$ and $x_i \neq y_i$.

<u>Notation</u>: <u>Superscripts</u> will be used to denote elements of a sequence. The reflexive, transitive closure of $\vdash$ is denoted $\overset{*}{\vdash}$. Hence $\underline{x \overset{*}{\vdash} y}$ if and only if there exist $x^0, x^1, x^2, \cdots, x^q$ $(q \geq 0)$ such that $x = x^0 \vdash x^1 \vdash x^2 \vdash \cdots \vdash x^q = y$. We write $\underline{x \vdash_p y}$ to denote a single application of production p to string x, while $\underline{x \vdash_1 y}$ means that exactly one production has been applied.

We are now ready to introduce the delay property into our asynchronous model.

<u>Definition 2.4</u>. Let $x^0, x^1, \cdots \in \Sigma^n$, $G = \langle \Sigma, P \rangle$ be an asynchronous grammar and $D \geq 0$ be an integer. The sequence $x^0, x^1, x^2, \cdots$ is a <u>D-computation</u> provided:

(i) $\forall j \geq 0$, $x^j \vdash x^{j+1}$ and

(ii) $\not\exists$ i, j such that $x_i^j = x_i^{j+k}$ and coordinate i is active in x^{j+k} for all $k = 0, 1, 2, \cdots, D+1$.

Hence a D-computation is a legal sequence of state transformations such that no active coordinate remains unchanged for $D + 1$ consecutive steps.

<u>Example 2.3</u>. Let $G_1 = \langle \{1,2\}, \{12 \longrightarrow 21\} \rangle$ then

$$\begin{array}{rl} 111222 & \vdash 112122 \\ & \vdash 121212 \\ & \vdash 212121 \\ & \vdash 221211 \\ & \vdash 222111 \end{array}$$

is a 0-computation for G_1, while

$$\begin{array}{rl} 111222 & \vdash 112122 \\ & \vdash 121\underline{12}2 \\ & \vdash 211\overline{21}2 \\ & \vdash 2121\underline{12} \\ & \vdash 2211\overline{21} \\ & \vdash 221211 \\ & \vdash 222111 \end{array}$$

is one of the 1-computations for G_1 on 111222.

Note that two active but idle transitions have been underscored.

Evidently, G_1 moves all 2's to the left and all 1's to the right in both the 0-computation and this particular 1-computation. Is this always the case for all D and all D-computations and how long does it take? We claim

(i) for any input of 1's and 2's, G_1 shifts all 2's to the left and all 1's to the right while preserving the total number of each for any 0-computation,

(ii) for all D, property (i) holds as well for all D-computations, and

(iii) any D-computation for G_1 halts in time less than $\delta(D+1)n$ for some constant $\delta \geq 1$.

The claim is quite intuitive but it is not simple to prove directly. For example, it is <u>not</u> the case even for a 0-computation, that once a 2 begins moving left, it continues to do so at a rate of at least $1/(D+1)$. This is because a 2 can "run into" a long sequence of 2's and be blocked (since the rule doesn't apply) for a long period of time. In short, it is quite possible for a 2 to exhibit a "hurry-up and wait" behavior.

Assertions (i) - (iii) are in fact true, but to prove them we employ some of the theory developed in the later sections. Our purpose in proving the assertions now is to underscore the proof strategy which we shall employ. We believe that it is an effective method of reducing the complexity of this type of proof and is, therefore, worthy of special emphasis.

The argument takes the following form:

(1) verify directly that the grammar computes the proper result for some convenient D-computation,

(2) find the time t required for the 0-computation,

(3) show that the grammar (or one equivalent to it) is elementary,[†]

(4) appeal to the CR theorem (Section 4) which says that all D-computations for elementary grammars compute the same result and the time is less than $(D+1)t$.

This strategy renders the proof of (i) - (iii) quite painless as we shall now see.

For correctness, it will be necessary to establish that the output is correct only for a single D-computation. Naturally, we choose D to simplify the proof and this is often the 0-computation or an ∞-computation, i.e. one where a specific sequence is chosen without regard for how long an active position is delayed. For the problem at hand, a specific 1-computation is most convenient. In particular, for any string x^0 of 1's and 2's, we choose the 1-computation with the property:

$x^0 \vdash x^1$ provided $12 \rightarrow 21$ is applied to all pairs

$x_j^0 x_{j+1}^0 = 12$ such that j is even.

$x^i \vdash x^{i+1}$ provided all active coordinates in x^i actually change

$(i > 0)$

This is clearly a 1-computation, since any $x_j^0 x_{j+1}^0 = 12$ where j is odd will be delayed one step; the computation is then synchronous thereafter.

[†] See Definition 2.8

This particular computation has been chosen because it has the following easily verified property:

(*) for $i > 0$, if i is odd (even) then $x_j^i \, x_{j+1}^i = 12$ changes to

$x_j^{i+1} \, x_{j+1}^{i+1} = 21$ iff j is odd (even).

Thus, during the synchronous portion of this computation the only active changes are to pairs with odd/even indices for odd computation steps and to pairs with even/odd indices for even computation steps. This property is important because it enables Floyd's theorem [5] on parallel sorting networks to be envoked: We interpret the synchronous portion of this computation as a parallel sort (into descending order) using interchanges. The interchanges of the sorting network are applied so that (*) holds. Floyd's theorem establishes that a vector of n elements sorted in parallel according to (*) halts with output in (descending) sorted order (all 2's to the left) in n steps or fewer. Since one step is required to "get into phase," this 1-computation halts with the proper result in $n+1$ steps. Thus, steps (1) and (2) are established. We find that (i) is true and that the 0-computation for it is time bounded by $n+1$. There is nothing to do for (4), so all that remains is to establish (3).

Elementary grammars (see Definition 2.8) have three properties: single change, persistence and determinacy.

Definition 2.5:. An asynchronous grammar $G = \langle \Sigma, P \rangle$ is single change iff $p \epsilon P$ implies p is of the form

$\alpha a \beta \rightarrow \alpha a' \beta$

where $\alpha, \beta \epsilon \Sigma^*$ and $a, a' \epsilon \Sigma$.

The single change property is fundamental since it enables a device to change independently of its neighbors. This is obviously not a property of G_1, but it can be modified to be single change.

<u>Example 2.5</u>. Let $G_2 = <\Sigma, P>$ where

$$\Sigma = \{1,2,A,B\} \text{ and}$$

$$\begin{aligned} P = \{&12 \rightarrow 1A \\ &1A \rightarrow BA \\ &BA \rightarrow B1 \\ &B1 \rightarrow 21 \\ &B2 \rightarrow 22\}. \end{aligned}$$

The intuition here is that the intermediate states, A and B, implement an "information passing" protocol where A means "a 2 is being sent left and acknowledgement of receipt is requested" and B means "the 2 has been received and is hereby acknowledged." Thus, the first four productions accomplish the 12 to 21 interchange. Production five is required because a 1 (placed by BA → B1) could have already been changed into a 2 due to the asynchronous execution.

Grammar G_2 computes the same result as G_1, but it is <u>not</u> true that its time satisfies that required in (iii) above. Indeed, the worst case behavior of G_2 is $\delta((D+1)n)^2$. The difficulty can be observed in the following example.

<u>Example 2.6</u>. Given G_2, a portion of a legal 1-computation for G_2 is

$$\begin{array}{rl} 111111222222 & \vdash 111111A22222 \\ & \vdash 11111BA22222 \\ & \vdash 11111B122222 \\ & \vdash 11111\underline{B1}A2222 \\ & \vdash 11111BBA2222 \\ & \quad\vdots \\ & \vdash 111112222221 \end{array}$$

The underscored transition did not "fire" and is now "locked out." In the worst case, it can remain so until a single 1 propagates all the way to the right and the B's change back into 2's. It will then change according to the last production. Apparently, the computational requirements are satisfied for G_2, but not the timing requirements. The solution, of course, is to add a new production.

Example 2.7. Let $G_3 = <\{1,2,A,B\}, P>$

$$P = \{12 \rightarrow 1A \\ 1A \rightarrow BA \\ BA \rightarrow B1 \\ B1 \rightarrow 21 \\ B2 \rightarrow 22 \\ BB \rightarrow 2B\}$$

The difficulty with G_2 is that it is not persistent.

Definition 2.6. An asynchronous grammar $G = <\Sigma, P>$ is persistent if

$\forall x,i,p$ and p'; $x \vdash_p y$ with $x_i \neq y_i$ and

$x \vdash_{p'} y'$ with $x_i = y'_i$

implies $\exists p'' \epsilon P$ such that $y' \vdash_{p''} y''$ with $y_i = y''_i$.

Persistence prevents an active transition from being "locked out" by the activity of its neighbors. Evidently, G_3 is persistent. G_3 also has the other property required of an elementary grammar, determinacy.

Definition 2.7. An asynchronous grammar $G = <\Sigma, P>$ is determinate if $\forall x$, i, p and p' and coordinate i active in x, if

$x \vdash_p y$ with $x_i \neq y_i$ and

$x \vdash_{p'} y'$ with $x_i \neq y'_i$

then $y_i = y'_i$.

Determinacy requires that there be a unique next state for any state change. Note that G_1 and G_2 are also determinate asynchronous grammars.

<u>Definition 2.8</u>. An asynchronous grammar is <u>elementary</u> if it is single change, determinate, and persistent.

Evidently G_3 is elementary, so we can appeal to the CR theorem, thus establishing (i) - (iii) for G_3. Generally, we will be working with single change grammars such as G_3 and hence we'd be finished. Since G_1 is not single change a few additional observations are required.

Clearly, for any D-computation of G_1 on x, each

$$x^i \vdash x^{i+1}$$

can be replaced by

$$x^i \vdash y^1 \vdash y^2 \vdash y^3 \vdash x^{i+1}$$

where y^1 has A's in all positions that change from 2 to 1 as $x^i \vdash x^{i+1}$, y^2 has B's to the left of the A's of y^1 and y^3 has 1's in A positions y^1. This is a legal 4D-computation for G_3. Hence (i) and (ii) must be true for G_1. Furthermore, if there is no δ such that all D-computations of G_1 are time bounded by $\delta(D+1)n$, no bound could exist for G_3. By the CR theorem it does exist and hence (iii) is established for G_1.

<u>Definition 2.9</u>. Let $G = \langle\Sigma, P\rangle$ be an asynchronous grammar and $x^0 \in \Sigma^n$. A D-computation $x^0, x^1, \cdots, x^m$ is said to <u>halt</u> if $\nexists\, x \in \Sigma^n$ such that $x^m \vdash x$. In this case x^m is called the <u>result</u> of the D-computation.

3. Synchronization Capabilities

In this section we demonstrate that asynchronous linear arrays cannot synchronize in any meaningful way. This will be done by showing that a problem, weaker than the "firing squad synchronization problem" cannot be solved. As noted previously, it is not surprising that the "standard version" of this problem cannot be solved, but it is not even possible for the two "soldiers" at the ends to "fire" at approximately the same time.

Definition 3.1. An asynchronous grammar $G = \langle \Sigma, P \rangle$ solves the $\langle f, D \rangle$ firing squad problem provided

(a) $\{g,c,q\} \subseteq \Sigma$, $\mathscr{S} \subseteq \Sigma$ and

(b) i. $\alpha \to \beta \in P$ implies $\alpha \notin \{q,c\}^*$

ii. for any D-computation $x^0, x^1, \cdots, x^m$ that halts (where $x^0 = gq^n c$), $|t_g - t_c| \le f(n)$

where

$$t_g = \min\{k \mid x_1^k \in \mathscr{S}\}$$
$$t_c = \min\{k \mid x_{n+2}^k \in \mathscr{S}\}$$

iii. All D-computations can be extended to halting D-computations.

Informally, the conditions can be viewed as follows: g = "general," q = "null state soldier," c = "colonel," and $\mathscr{S}$ = the set of "shoot" states; g and c mark the ends of the array. Condition (bi) requires that the process be initiated by the "general," (bii) requires that the firing times of the "general" and "colonel" be within $f(n)$ of each other. Note that the definition of t_g and t_c requires that only the first "bullet" fired by the general, and the first "bullet" fired by the colonel are of interest. Thus, the classic solution solves the $\langle 0,0\rangle$ firing squad problem, i.e. the problem is solved synchronously with the general and colonel (and all other soldiers) firing at the same time.

Theorem 3.1. Suppose that the asynchronous grammar $G = \langle\Sigma, P\rangle$ solves the $\langle f,D\rangle$ firing squad problem with $D > 0$. Then there exists a constant $\delta > 0$ such that $f(n) \geq \delta n$.

Proof. Let G be given as required by Definition 3.1, and $x^0,x^1,\cdots,x^m$ be a halting 0-computation where $x^0 \in \{gq^*c\}$, $|x^0| = n+2$ and

$$t_g = \min\{k|x_1^k \in \mathcal{F}\}$$

$$t_c = \min\{k|x_{n+2}^k \in \mathcal{F}\}.$$

For each $p \in P$, writing p as $\alpha a\gamma \longrightarrow \alpha' b\gamma$ with $a \neq b$ indicates the rightmost changed symbol in p. Then let $w_R = \max_P\{|\alpha'||p \in P\}$.

Similarly writing p as $\alpha a\gamma \longrightarrow ab\gamma'$ with $a \neq b$ indicates the leftmost change in p. Let $w_L = \max_P\{|\gamma'||p \in P\}$.

Finally, we define w as $w = \max\{w_L,w_R\}$.

That is, w is the maximum distance over which a single production application can cause a change. Therefore, as with the classical firing squad problem, no signal can travel from g to c in less than n/w steps; thus $t_c \geq n/w$. Now, suppose that $f(n) \leq n/2w$ since if it were not, the theorem is true. But, $t_g \geq n/2w$ since otherwise $|t_c - t_g| > n/2w$. Therefore, both t_g and t_c are at least $n/2w$.

Suppose $t_g \leq t_c$ (the other case is analgous). Without loss of generality, let $n/4w$ be an integer. Define $k = t_g - n/4w$. Now, there exists a 1-computation

$$x^0,x^1,\cdots,x^k, y^{k+1},\cdots,y^{k+s}$$

with the property that $y_1, y_2, \cdots, y_{n/2}$ fire at each step (if possible) and $y_{n/2+1} \cdots y_{n+2}$ fire (if possible) at every second step.

Then the following are true:

(i) $x_j^{k+1} = y_j^{k+1}$ $\quad j = 1, 2, \cdots, \frac{n}{2} - w$ and, in general

(ii) $x_j^{k+i} = y_j^{k+i}$ $\quad j = 1, 2, \cdots, \frac{n}{2} - iw$.

Note that here x_j^{k+i} refer to the 0-computation and y_j^{k+i} refer to the 1-computation. For $i = 1$, by definition of the 1-computation all transitions on $y_1^{k+1} \cdots y_{\frac{n}{2}-w}^{k+1}$ take place exactly as for the 0-computation.

Thus (i) follows. For $i > 1$ a simple induction proves (ii).

When $i = n/4w$, then $x_j^{t_g} = y_j^{t_g}$ on machines $j = 1, \cdots, n/4$. Hence $t_g = t'_g$ where

$$t'_g = \min\{\ell \mid y_1^{\ell} \in \mathcal{S}\}$$

and the "general" fires at the same step in this particular 1-computation as in the 0-computation. Now, analyzing when the "colonel" fires, we claim that for $i = 1, 2, \cdots, n/4w$

(iii) $x_j^{k+i} = y_j^{2k+2i}$ for $j = n/2 + 1 + iw, \cdots, n+2$.

Of course, by construction, $y_j^{2k+2i} = y_j^{2k+2i-1}$ for $j = n/2 + 1, \cdots, n + 2$. Clearly, (iii) holds also by induction. Now, when $i = n/4w - 1$, then

$$x_j^{t_g-1} = y_j^{2t_g-2} \quad \text{for } j = n/2 + 1 + n/4 - w, \cdots, n + 2.$$

In particular, for large enough n, $x_{n+2}^{t_g-1} = y_{n+2}^{2t_g-2}$. Define

$$t'_c = \min\{\ell \mid y_{n+2}^{\ell} \in \mathcal{S}\}$$

Since $t_c \geq t_g$, $x_{n+2}^{t_g-1} \notin \mathcal{S}$. Therefore $y_{n+2}^{2t_g-2} \notin \mathcal{S}$.

Consequently, $t'_c > 2t_g - 2$. Thus

$$|t'_c - t_g| > t_g - 2, \text{ so}$$

$$|t'_c - t'_g| > t_g - 2.$$

It follows that

$$f(n) \geq n/2w - 1. \quad \square$$

We note that although our theorem states only that $f(n) \geq \delta n$ we have actually proven a somewhat stronger result, namely that $f(n) \geq n/2w - 1$. The sharpness of this result is particularly noticeable for the case of an ordinary array of machines with $|\alpha| = 3$ mentioned earlier. In this case $w = 1$ so our result becomes $f(n) \geq n/2 - 1$. However, such a single change grammar can readily be devised to "find the middle" of an array. Briefly, this can be accomplished by first getting the array into a condition of "alternating 1's and 2's" in machines, then applying grammar G_3 of Example 2.7 to move all 2's to the left and 1's to the right. Finally, the 21 boundary indicates the "middle." Now, firing from the middle outward gives a firing for which $f(n) \leq nD/2$. The worst case being that one half fires at the fastest possible rate and the other half fires at the slowest possible rate. Thus, for $D = 1$ we get the very tight lower and upper bounds of $n/2 - 1 \leq f(n) \leq n/2$. Although it seems clear that D should enter the lower bound in a multiplicative way, so as to give fairly tight lower and upper bounds for any D, it is not immediately apparent how D could be appropriately introduced into the proof.

Theorem 3.1 can be viewed as being sharp in still another sense. Given any D and any $\epsilon > 0$ there is an asynchronous grammar that solves the $\langle f,D \rangle$ firing squad problem with $f(n) = \epsilon n$. The solution is just to have the general send a signal down to the colonel. This signal causes each machine to fire as it receives it. Since one can clearly send a signal from the general to the colonel in $\leq \epsilon n$ time by using productions of the form $sq^k \to s^{k+1}$ (s = shoot state, q = quiescent state) for k large, it follows that one can solve the $\langle f,D \rangle$ firing squad problem with $f(n) = \epsilon n$.

4. The CR Theorem

The objective of this section is to prove the CR theorem which states that all D-computations for a given input and elementary grammar yield the same result and the time required is less than or equal to (D+1) times the 0-computation time for that input. Several preliminaries are required prior to the statement and proof of the theorem.

This entire section implicitly refers to an elementary grammar $G = <\Sigma, P>$.

Definition 4.1. Let $x \in \Sigma^n$ and $1 \leq k \leq n$. Define $f_k(x)$, a substitution function as

$$f_k(x) = \begin{cases} y\alpha b\beta z & \text{if } \exists \alpha a\beta \rightarrow \alpha b\beta \in P \text{ such that} \\ & x = y\alpha a\beta z \text{ and } |y\alpha a| = k, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Since G is elementary, it is determinate and hence $f_k(x)$ is well defined. We next establish that for a single transition, the order of application of the productions is immaterial. Notationally, if f, g, h are substitution functions for x, let fgh(x) denote the repeated function application

$$f(g(h(x))).$$

A A simple consequence of the definition of $\vdash$ and the substitution functions is given in the first lemma.

Lemma 4.1. $x \vdash y$ if $y = f_{d_1} \cdots f_{d_q}(x)$ where $f_{d_i}(x)$ is defined, $1 \leq i \leq q$.

Lemma 4.2. Let $x \in \Sigma^n$ and $f_{d_i}(x)$, $1 \le i \le q$ be a set of substitution functions defined for x. Then for any permutation π on $\{1,\cdots,q\}$,

$$f_{d_{\pi_1}} \cdots f_{d_{\pi_q}}(x) = f_{d_1} \cdots f_{d_q}(x) = y \in \Sigma^n$$

Proof. If $f_r(x)$ and $f_s(x)$ are defined, then $r \neq s$ implies

$$f_r f_s(x) = f_s f_r(x) = z \in \Sigma^n,$$

since G is determinate and persistent. A straightforward induction on the number of interchanges required to reorder π to $1\cdots q$ completes the proof. □

Suppose f_{d_i}, $1 \le i \le r$ is the set of all substitution functions defined for x, then we define $x \vdash_{\text{par}} y$ as $y = f_{d_1} \cdots f_{d_r}(x)$. We also note that $x \vdash_1 x'$ if $x' = f_{d_k}(x)$ for some k, $1 \le k \le r$.

The next lemma, illustrated in Figure 4.1a, will be an integral part of the induction of the subsequent lemma.

Lemma 4.3. If $x \in \Sigma^n$, $x \vdash_1 y$, $x \vdash_{\text{par}} x'$, $y \vdash_{\text{par}} y'$ imply $x' \overset{*}{\vdash_1} y'$.

Proof. Let $f_{d_i}(x)$ be all defined substitution functions for x, $1 \le i \le r$. From the definition of $\vdash_{\text{par}}$, and Lemma 4.2

$$x' = f_{d_1} \cdots f_{d_r}(x) \qquad (1)$$

and

$$y = f_{d_r}(x)$$

with renaming if necessary. Let $f_{c_i}(y)$ $1 \le i \le q$ be all defined functions for y. Then

$$y' = f_{c_1} \cdots f_{c_q} f_{d_r}(x).$$

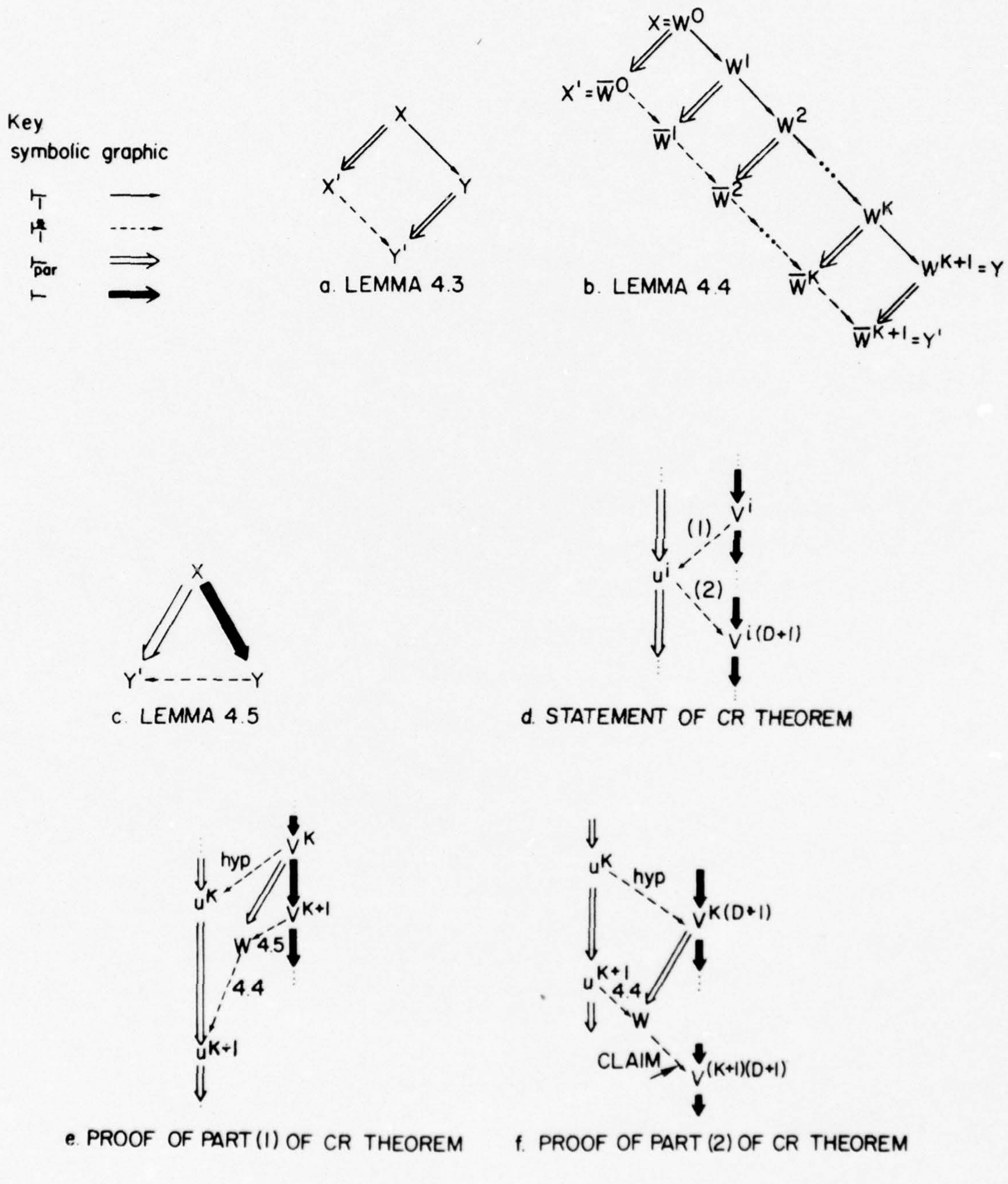

Key
symbolic graphic
a. LEMMA 4.3
b. LEMMA 4.4
c. LEMMA 4.5
d. STATEMENT OF CR THEOREM
e. PROOF OF PART (1) OF CR THEOREM
f. PROOF OF PART (2) OF CR THEOREM
hyp
4.5
4.4
CLAIM
(1)
(2)

Figure 4.1 Graphic representation of selected lemmas and theorems.

By persistence $f_{d_i} f_{d_r}(x)$ is defined for all i, $1 \le i \le r - 1$, and so $\{f_{d_1}, \cdots, f_{d_{r-1}}\} \subseteq \{f_{c_1}, \cdots, f_{c_q}\}$. By Lemma 4.1 then we can reorder and substitute to get

$$y' = f_{e_1} \cdots f_{e_s} f_{d_1} \cdots f_{d_r}(x) \qquad (2)$$

for some f_{e_i} where

$$\{f_{e_1}, \cdots, f_{e_s}\} = \{f_{c_1}, \cdots, f_{c_q}\} - \{f_{d_1}, \cdots, f_{d_{r-1}}\}.$$

Combining (1) and (2) and using the definition of $\vdash_{\overline{1}}$, it follows that $x' \overset{*}{\vdash_{\overline{1}}} y'$. □

Lemma 4.3 is generalized in Lemma 4.4. The lemma and proof are illustrated by Figure 4.1b.

Lemma 4.4. If $x \in \Sigma^n$, $x \overset{*}{\vdash_{\overline{1}}} y$, $x \vdash_{\overline{par}} x'$, $y \vdash_{\overline{par}} y'$ then $x' \overset{*}{\vdash_{\overline{1}}} y'$.

Proof. We use induction on the number of applications of $\vdash_{\overline{1}}$ in going from x to y. If $x = y$ the result is trivial and when the length is 1, the result follows from Lemma 4.3. Let

$$x = w^0 \vdash_{\overline{1}} w^1 \vdash_{\overline{1}} \cdots \vdash_{\overline{1}} w^k \vdash_{\overline{1}} w^{k+1} = y$$

$k \ge 2$. Let

$$w^i \vdash_{\overline{par}} \bar{w}^i \qquad 0 \le i \le k+1$$

By Lemma 4.3,

$$\bar{w}^i \overset{*}{\vdash_{\overline{1}}} \bar{w}^{i+1} \qquad 0 \le i \le k$$

Thus,

$$\bar{w}^1 \overset{*}{\vdash_{\overline{1}}} \bar{w}^{k+1}$$

and since $x' = \bar{w}^1$ and $y' = \bar{w}^{k+1}$ it follows that $x' \overset{*}{\overline{1}} y'$. □

Figure 4.1c illustrates the next lemma.

Lemma 4.5. If $x \in \Sigma^n$, $x \vdash y$ and $x \vdash_{par} y'$ then $y \vdash_1^* y'$.

Proof. Let $f_{d_i}(x)$ be all defined substitution functions for x, $1 \le i \le r$.

Then

$$y = f_{d_j} \cdots f_{d_r}(x)$$

for some $j \le r$ by Lemma 4.1 and (possibly) renaming.

But

$$y' = f_{d_1} \cdots f_{d_r}(x) = f_{d_1} \cdots f_{d_{j-1}}(y)$$

and hence $y \vdash_1^* y'$. □

We can now state the main result of this section.

CR Theorem. Let $G = <\Sigma, P>$ be an elementary asynchronous grammar and $x \in \Sigma^n$. Let $x = u^0, u^1, u^2, \cdots, u^\ell$ be a halting 0-computation of G on x and $x = v^0, v^1, v^2, \cdots, v^m$ be any halting D-computation of G on x. Then

(1) $v^i \vdash_1^* u^i$ and

(2) $u^i \vdash_1^* v^{i(D+1)}$

$0 \le i \le \ell$.

Note: If $m < \ell(D+1)$ we suppose the D-computation has been extended such that $m < p \le \ell(D+1)$ implies $v^p = v^m$. This theorem and its proof are illustrated in Figure 4.1d - 4.1f.

Proof.

(1) We prove this by induction on i. For $i = 0$, it is trivially true. By hypothesis $v^i \vdash_1^* u^i$ for $i \le k$. By definition of 0-computation $u^j \vdash_{par} u^{j+1}$ $0 \le j < \ell$. Define $v^k \vdash_{par} w$

for some w. By Lemma 4.4, $w \vdash_1^* u^{k+1}$. But, by Lemma 4.5 $v^{k+1} \vdash_1^* w$. Therefore $v^{k+1} \vdash_1^* u^{k+1}$.

(2) This is also proved by induction on i. For $i = 0$, the result is trivial. By induction hypothesis, $u^i \vdash_1^* v^{i(D+1)}$, for $i \leq k$, and by definition of 0-computation $u^j \vdash_{par} u^{j+1}$ for $0 \leq j < \ell$.

Claim. $\exists w$ such that $v^{k(D+1)} \vdash_{par} w \vdash_1^* v^{k(D+1)+D+1}$. (We prove the claim in a moment.) By Lemma 4.4 $u^{k+1} \vdash_1^* w$ and therefore $u^{k+1} \vdash_1^* v^{k(D+1)+D+1}$.

To see that the claim is true, let $f_{d_i}(v^{k(D+1)})$ be all defined functions, then

$$w = f_{d_1} \cdots f_{d_q}(v^{k(D+1)})$$

by definition of $\vdash_{par}$. For $1 \leq j \leq D + 1$ define $f_{c_1^j}, \cdots, f_{c_{q_j}^j}$ to be all defined substitution functions such that

$$v^{k(D+1)+j} = f_{c_1^j} \cdots f_{c_{q_j}^j}(v^{k(D+1)+j-1}).$$

But by persistence, the definition of D-computation and a reordering argument similar to that of Lemma 4.2, $\exists f_{e_1}, \cdots, f_{e_r}$ such that

$$v^{k(D+1)+D+1} = f_{e_1} \cdots f_{e_r} f_{d_1} \cdots f_{d_q}(v^{k(D+1)})$$

and the claim follows. □

The import of the CR Theorem can be seen in the following corollaries.

Corollary 4.1. Let $G = \langle\Sigma, P\rangle$ be an elementary asynchronous grammar and $x \in \Sigma^n$. If $y \in \Sigma^n$ is the result of any halting D-computation of G on x, then y is the result of all halting D-computations of G on x.

Corollary 4.2. Let $G = \langle\Sigma, P\rangle$ be an elementary asynchronous grammar and $x \in \Sigma^n$. If a 0-computation of G on x halts in ℓ steps, then any D-computation of G on x halts in less than or equal to $\ell(D+1)$ steps.

It is important to note that the requirements of determinacy and persistence are necessary in the sense that the CR Theorem is false if they are eliminated. This is clear for determinacy. Grammar G_2 from Section 2, which is not persistent and which executed in approximately n^2 rather than n steps, demonstrates this for persistence. We shall have occasion to use these two corollaries in the next section on recognition capabilities.

The connection between asynchronous computation and the Church-Rosser property has been observed before by several researchers [6, 8]. The contribution here is that in the presence of bounded delay $(D < \infty)$ not only do asynchronous computations "behave the same" but they operate in "about the same time." In particular Corollary 4.2 is new, while Corollary 4.1 for $D = \infty$ can be proved fairly directly from results in [6] and [8].

5. Recognition Properties of Linear Asynchronous Grammars.

The goal of this section is to argue that the sets recognizable by cellular 1-dimensional arrays [10] in time t can also be recognized by linear asynchronous grammars in time 3(D+1)t for all delays $D \geq 0$. Since cellular arrays can solve a wide class of recognition problems in a synchronous and efficient manner, we can conclude that these problems can also be performed asynchronously without serious time degradation.

The overall strategy begins by noting that any single change, determinate asynchronous grammar (of which cellular arrays are a special case) can be put into a normal form with certain properties. The next step is to show how to construct a persistent grammar from the normal form grammar, such that both grammars produce the same output in the synchronous case (0-computations). Finally, we appeal to the CR Theorem to establish that for any single change, determinate asynchronous grammar, there exists an elementary grammar accepting the same set in time 3(D+1)t, where t is the recognition time for a 0-computation of the original grammar. The desired result then follows as a corollary since cellular 1-dimensional arrays correspond to single change, determinate asynchronous grammars.

To simplify the exposition, we omit two details. First, we omit the construction of the normal form and, secondly, we ignor the details involving the end-points of the array. For convenience, the reader can suppose that the configurations are bounded by end markers and appropriate productions exist for handling the markers. The general case is unaffected by this assumption.

Let $G = \langle\Sigma, P\rangle$ be any single change, determinate asynchronous grammar. (Note that cellular 1-dimensional arrays satisfy this requirement.)

Definition 5.1. A single change, determinate asynchronous grammar $G_N = <\Sigma, P_N>$ is a normal form for G, if there exists a k such that

(i) $\forall\alpha \longrightarrow \beta \in P_N, \ |\alpha| = k$

(ii) $a_1\cdots a_k \longrightarrow b_1\cdots b_k \in P_N$ implies $a_1\cdots a_{\lfloor k/2\rfloor} = b_1\cdots b_{\lfloor k/2\rfloor}$ and

$$a_{\lfloor (k+1)/2\rfloor+1}\cdots a_k = b_{\lfloor (k+1)/2\rfloor+1}\cdots b_k$$

(iii) $\alpha \longrightarrow \beta \in P_N \iff \exists p \in P$ such that $\alpha \vdash_{\overline{p}} \beta$.

Informally, requirement (i) states that all productions are of the same size, while (ii) guarantees that the modification is to the "middle" term in the production. This latter requirement implies, of course, that k is an odd integer. Property (iii) requires the same behavior from the two production sets on k length strings. It can be seen that cellular 1-dimensional arrays are represented by normal form grammars with $k = 3$. Let G_N be a normal form for G and let $m = (k+1)/2$ in the sequel.

Lemma 5.1. For any G, there exists a G_N.

Proof. This can be done by suitably padding out productions of the original grammar. The construction is omitted. □

Lemma 5.2. If G is single change and determinate, then G_N is single change and determinate.

Proof. The single change property follows directly from Definition 5.1 (ii) and the determinacy property follows from Definition 5.1 (iii). □

Lemma 5.3. Let $x^0,\cdots,x^h$ be a 0-computation for G and $x^0 = y^0,\cdots,y^h$ be a 0-computation for G_N on x^0. Then $x^\ell = y^\ell \quad 0 \le \ell \le h$.

Proof. By induction on ℓ. For $\ell = 0$ the result is immediate. Suppose it is true for all $i \le \ell$. If $x_j^{\ell} = x_j^{\ell+1}$, no $p \in P$ applies at j in x^{ℓ}. If $p_N \in P_N$ applies at j in y^{ℓ}, requirement (iii) is contradicted, so $x_j^{\ell+1} = y_j^{\ell+1}$. If $x_j^{\ell} \neq x_j^{\ell+1}$, some $p \in P$ applies at j in x^{ℓ}. Let $\alpha = x_{j-m+1}^{\ell} \cdots x_{j+m-1}^{\ell}$. Then $\alpha \vdash_{p} \beta$ and by (iii), $\alpha \to \beta \in P_N$ implying j active in y^{ℓ}. Thus $x_j^{\ell+1} = y_j^{\ell+1}$. □

Having found a normal form for G, we now seek to construct a new grammar G' which is persistent. We first require a definition.

Definition 5.2. Let $G_N = <\Sigma, P_N>$ be a single change, determinate asynchronous grammar in normal form. The completion of G_N is a system $G_c = <\Sigma, P_c>$ such that $P_c = P_N \cup \{\alpha \to \alpha \mid \nexists \beta, \alpha \to \beta \in P_N\}$. Informally, the completion of G_N has the production set as G_N with all "idling" productions added. Thus, the completion is not an asynchronous grammar. This is no problem since the completion will be transformed into a legal asynchroncus grammar below.

A few comments are in order about the forthcoming construction. The goal is to achieve persistence. The technique by which this is accomplished is to define a protocol that enables each device to acquire inputs from its neighbors. The protocol is basically three fold: (1) a device announces its intention to change state. At this point, every neighbor that depends upon the device's current value for their next state change must now retrieve the input. This is done by having the neighbors announce their intent to change state. When all the neighbors have announced, (2) a device is allowed to perform its transition. After the device and its neighbors have changed state, (3) they acknowledge that fact by becoming quiescent. A new cycle is then ready to begin. The purpose, therefore, for completing the grammar is to enable transitions that wouldn't otherwise fire to receive input, even though no new state change will result.

Definition 5.3. Given $G_c = <\Sigma, P_c>$ the completion of an asynchronous grammar, define the alphabet sets

$$\Sigma^a = \{[{a \atop b}] \mid a_1 \cdots a_{m-1} a a_{m+1} \cdots a_k \to a_1 \cdots a_{m-1} b a_{m+1} \cdots a_k \epsilon P_c\}$$

$$\Sigma_a = \{[{b \atop -}] \mid a_1 \cdots a_{m-1} a a_{m+1} \cdots a_k \to a_1 \cdots a_{m-1} b a_{m+1} \cdots a_k \epsilon P_c\}$$

$$\Sigma_{a*} = \{b \mid [{b \atop -}] \in \Sigma_a\}.$$

Informally, each element of Σ^a is a state a device enters when it announces its intent to change from state a to state b. Each element of Σ_a is a transition state it enters before becoming quiescent. Each element of Σ_{a*} is a quiescent state. We next define the productions for the new grammar from the completion.

Definition 5.4. Given the completion $G_c = <\Sigma, P_c>$ of a normal form grammar define the production sets

$$P_a =$$

$$\{c_1 \cdots c_{m-1} a_m c_{m+1} \cdots c_k \to c_1 \cdots c_{m-1} [{a_m \atop b_m}] c_{m+1} \cdots c_k$$

$$\mid a_1 \cdots a_m \cdots a_k \to a_1 \cdots b_m \cdots a_k \in P_c \text{ and}$$

$$c_i \in \Sigma^{a_i} \cup \{a_i\} \quad 1 \le i \le k \text{ and } i \ne m\},$$

$P_t =$

$$\{d_1 \cdots d_{m-1}\left[{a_m \atop b_m}\right]d_{m+1} \cdots d_k \rightarrow d_1 \cdots d_{m-1}\left[{b_m \atop -}\right]d_{m+1} \cdots d_k$$

$$|\, a_1 \cdots a'_m \cdots a_k \rightarrow a_1 \cdots b_m \cdots a_k \in P_c \text{ and}$$

$$d_i \in \Sigma^{a_i} \cup \Sigma_{a_i}, \quad 1 \leq i \leq k \text{ and } i \neq m\},$$

$P_q =$

$$\{e_1 \cdots e_{m-1}\left[{b_m \atop -}\right]e_{m+1} \cdots e_k \rightarrow e_1 \cdots e_{m-1} b_m e_{m+1} \cdots e_k$$

$$|\, a_1 \cdots a_m \cdots a_k \rightarrow a_1 \cdots b_m \cdots a_k \in P_c \text{ and}$$

$$e_i \in \Sigma_{a_i} \cup \Sigma_{(a_i)^*}, \; 1 \leq i \leq k, \; i \neq m\}.$$

The P_a productions accomplish the announcing task. Note that the neighbors may or may not have announced when a given position does so. The P_t productions perform the transition and they require that the neighbors have either performed the transition or announced. The P_q productions return to quiescent state where the neighbors have either done so as well or at worst, they have performed their transition.

Let $\Sigma' = \bigcup_{a \in \Sigma} \Sigma^a \cup \Sigma_a$ in the sequel.

<u>Lemma 5.4.</u> $G' = \langle \Sigma \cup \Sigma', P_a \cup P_t \cup P_q \rangle$ is a single change asynchronous grammar.

In the sequel we will use G' for this grammar.

<u>Proof</u>. Immediate, by construction. □

<u>Lemma 5.5</u>. G' is determinate.

<u>Proof</u>. Determinacy is vacuously satisfied since no two productions have the same left hand side and all productions change only the middle symbol. □

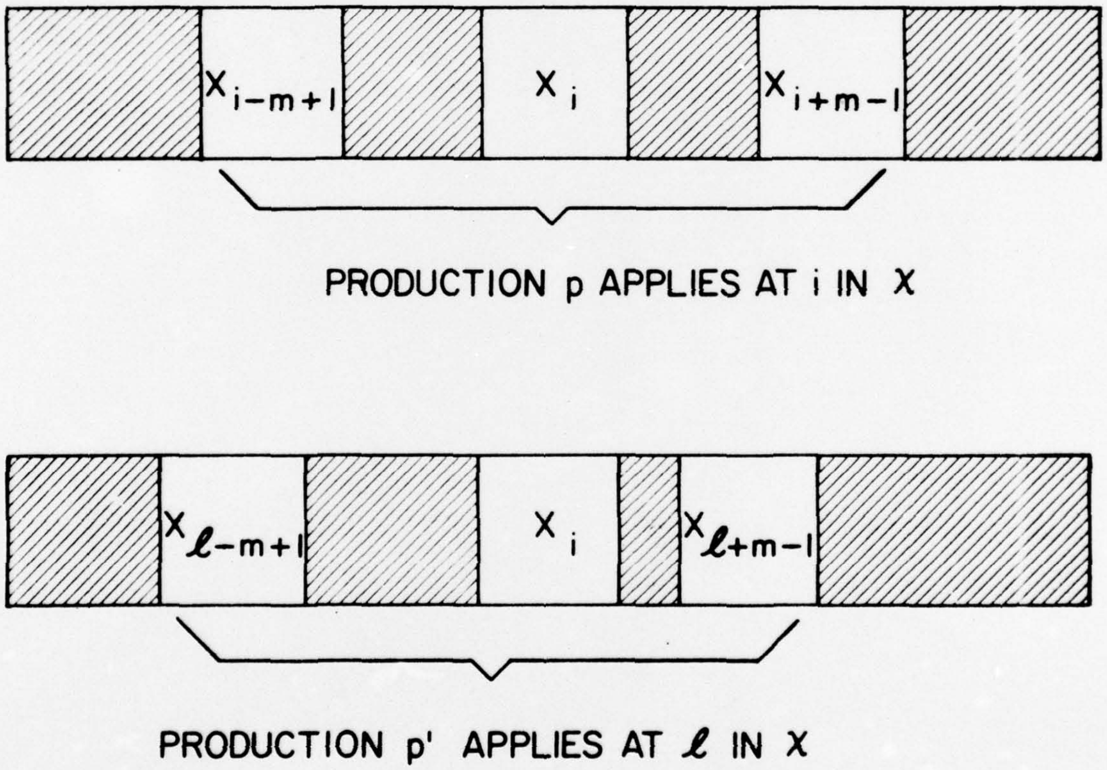

X_{i-m+1}
X_i
X_{i+m-1}
PRODUCTION p APPLIES AT i IN X
$X_{\ell-m+1}$
X_i
$X_{\ell+m-1}$
PRODUCTION p' APPLIES AT ℓ IN X

Figure 5.1. Production application with overlap.

Lemma 5.6. G' is persistent.

Proof. Let $x \vdash_{p} y$ for some $p \in P_a \cup P_t \cup P_q$ such that $x_i \neq y_i$ and $x \vdash_{p'} y'$ for some $p' \in P_a \cup P_t \cup P_q$ such that $x_i = y'_i$. Assume, in contradiction to the persistence requirement, that $\forall p'' \in P_a \cup P_t \cup P_q$ such that $y' \vdash_{p''} y''$ implies $y''_i \neq y_i$. Let $p = u_1 \cdots u_m \cdots u_k \to u_1 \cdots u'_m \cdots u_k$ and $p' = v_1 \cdots v_m \cdots v_k \to v_1 \cdots v'_m \cdots v_k$. Thus $x \vdash_{p} y$ implies

$$x_{i-m+1} \cdots x_{i+m-1} = u_1 \cdots u_k$$

and $x \vdash_{p'} y'$ implies

$$x_{\ell-m+1} \cdots x_{\ell+m-1} = v_1 \cdots v_k \quad \text{for some } \ell \text{ with } \quad i-m+1 \leq \ell \leq i+m-1$$

since otherwise p would still be active at i in x. (See Figure 5.1) Moreover, since $x_i = y'_i$, $i \neq \ell$. Define $j = i-\ell$, then

(*) $\quad x_i = u_m = v_{m+j} \quad$ and

(**) $\quad x_\ell = v_m = u_{m-j}$

Case 1. ($p \in P_a$). By construction, $p \in P_a$ implies $u_m \in \Sigma$. But (*) implies $v_{m+j} \in \Sigma$, so $p' \notin P_t$. If $p' \in P_q$, then the construction implies $v_m \in \Sigma_a$ for some a, but (**) implies $u_{m-j} \in \Sigma_a$ contradicting $p \in P_a$. Therefore $p' \in P_a$, and $v'_m \in \Sigma^a$ for some a. But by construction, a production $w_1 \cdots w_m \cdots w_k \to w_1 \cdots w'_m \cdots w_k$ exists such that $1 \leq z \leq k$, $z \neq m-j$, $u_z = w_z$, and $w_{m-j} = v'_m \in \Sigma^a$. Moreover, $w'_m = u'_m$. Thus, the assumption is false if $p \in P_a$.

Case 2. ($p \in P_t$). By construction $p \in P_t$ implies $u_m \in \Sigma^a$ for some a. But (*) implies $v_{m+j} \in \Sigma^a$, so $p' \notin P_q$. If $p' \in P_a$, then the construction implies $v_m \in \Sigma$, but (**) implies $u_{m-j} \in \Sigma$ contradicting

$p \in P_t$. Hence, $p' \in P_t$, and $v'_m \in \Sigma_a$ for some a. But by construction, a production $w_1 \cdots w_m \cdots w_k \rightarrow w_1 \cdots w'_m \cdots w_k$ exists such that $1 \le z \le k$, $z \ne m-j$, $u_z = w_z$ and $w_{m-j} = v'_m \in \Sigma_a$. Moreover, $w'_m = u'_m$. Thus the assumption is false, if $p \in P_t$.

Case 3. $(p \in P_q)$. By construction, $p \in P_q$ implies $u_m \in \Sigma_a$ for some a. But (*) implies $v_{m+j} \in \Sigma_a$, so $p' \notin P_a$. If $p' \in P_t$, then the construction implies $v_m \in \Sigma^a$, for some a, but (**) implies $u_{m-j} \in \Sigma^a$ contradicting $p \in P_q$. Hence, $p' \in P_q$. An argument similar to the above guarantees a production, so the assumption is false if $p \in P_q$. □

Lemma 5.7. The sequence $x^0, x^1, \cdots, x^{3h}$ is a 0-computation sequence for G' on x^0 iff $x^0 = y^0, y^1, \cdots, y^h$ is a 0-computation sequence for G_N on x^0, and $y^i = x^{3i}$ $0 \le i \le h$.

Proof. Clearly, by construction

$$x^{3i} \in \Sigma^n \qquad i = 0,1,2,\cdots,3h$$

$$x^{3i+1} \in \{\bigcup_{a\in\Sigma} \Sigma^a\}^n \qquad i = 0,1,\cdots,3(h-1)$$

$$x^{3i+2} \in \{\bigcup_{a\in\Sigma} \Sigma_a\}^n \qquad i = 0,1,\cdots,3(h-1)$$

since $x^0, x^1, \cdots, x^{3h}$ is a 0-computation. Suppose $x^0 \overset{*}{\vdash} x^{3i}$ satisfies the lemma for $i = 0,1,\cdots,\ell$. To prove the if part, let $p_1 \in P_a$ apply at j in $x^{3\ell}$, then $x^{3\ell} \vdash_{p_1} x^{3\ell+1}$ and $x_j^{3\ell+1} = [{a \atop b}]$ for some $a,b \in \Sigma$. Each of the $m-1$ neighbors on either side is in a state chosen from $\bigcup_{a\in\Sigma} \Sigma^a$ as well. Hence there is $p_2 \in P_t$ such that $x^{3\ell+1} \vdash_{p_2} x^{3\ell+2}$ and $x_j^{3\ell+2} = [{b \atop \_}]$. Now, the $m-1$ neighbors on either side are in a state chosen from $\bigcup_{a\in\Sigma} \Sigma_a$. Hence some $p_3 \in P_q$ applies and $x^{3\ell+2} \vdash_{p_3} x^{3(\ell+1)}$ such that

$x_j^{3(\ell+1)} = b$. But, by construction, these three transitions imply there exist $p \in P_c$ such that $x^{3\ell} \vdash_p x^{3(\ell+1)}$ with $x_j^{3(\ell+1)} = b$. If $a \neq b$ then $p \in P_N$ and so $x_j^{3(\ell+1)} = y_j^{\ell}$. If $a = b$, no production in P_N applies (p was added by completion) and $y_j^{\ell} = y_j^{\ell+1}$ at the $(\ell+1)^{st}$ step of the 0-computation of G_N on x^0.

To prove the only if part, suppose in the 0-computation of G_N on x^0, $y^{\ell} \vdash y^{\ell+1}$. If p is active at j in y^{ℓ}, $p \in P_c$. If $y_j^{\ell} = y_j^{\ell+1}$, a completion production has been added. In either case, according to the construction a production $p_1 \in P_a$ exists such that $x^{3\ell} \vdash_{p_1} x^{3\ell+1}$ with $x_j^{3\ell+1} = [{a \atop b}]$ and $y_j^{\ell} = a$ and $y_j^{\ell+1} = b$. It is easily observed that other productions exist such that $x^{3\ell+1} \vdash x^{3\ell+2} \vdash x^{3(\ell+1)}$ and $x_j^{3(\ell+1)} = b = y_j^{\ell+1}$. □

Theorem 5.1. Given a single change, determinate asynchronous grammar $G = \langle\Sigma, P\rangle$ there exists an elementary grammar $G' = \langle\Sigma', P'\rangle$ such that $\forall n$ and $\forall D > 0$, if $x^0 \in \Sigma^n$ and $x^0,\cdots,x^{h_1}$ is halting 0-computation of G on x^0, then $\exists h_2$ such that

(i) $x^0,\cdots,x^{h_2}$ is a halting D-computation for G' on x^0 with $x^{h_2} = x^{h_1}$, and

(ii) $h_2 \leq 3(D+1)h_1$.

Proof. Form G_N, the normal form for G. Complete the normal form and construct G'. The 0-computation for these grammars yield the same output with G' operating at most 3 times slower than G. By the CR theorem of the previous section, G' satisfies (i) and (ii) since it is elementary, by Lemmas 5.4, 5.5, and 5.6. □

Corollary 5.1. A set recognized by a cellular 1-dimensional array in time t is recognized by some elementary grammar, asynchronously, in time less than or equal to $3(D+1)t$, for all $D \geq 0$.

Proof. Let $\delta: \Sigma \times \Sigma \times \Sigma \longrightarrow \Sigma$ be the transition function for the cellular 1-dimensional array [10]. Then define $G = <\Sigma, P>$ where $a_1a_2a_3 \longrightarrow a_1b_2a_3 \in P$ iff $\delta(a_1,a_2,a_3) = b_2$, with $a_2 \neq b_2$, $\forall a_1,a_2,a_3,b_2 \in \Sigma$ □

6. Conclusions

What we have done in this paper is to introduce the notion of bounded delay asynchronism and to study some of its properties in terms of a very special structure which we called linear asynchronous structures. We have shown three main results. First, that these structures cannot be synchronized well; the "gap between firings" is a function of the number of elements in the structure. Second, that under suitable hypotheses (elementary grammars) the systems compute unique values, and are not much "slower" than synchronous structures, and finally that these systems are computationally as powerful as synchronous systems.

Several natural questions arise. The first is: What happens for more complicated structures? Natural extensions would be to higher dimensional uniform structures and to tree and graph structures in which each node had the same in and out degree. It would seem that similar results could be obtained.

The second question arises from our CR Theorem in Section 4. We have already shown in Section 4 how there is an intimate connection between Church-Rosser systems and linear asynchronous structures. But our CR Theorem says something more than the usual Church-Rosser type result by introducing timing, or number of steps, comparisons between the "shortest" and "longest" paths to the unique result. The question, then, is how or when can such timing results be obtained for other types of Church-Rosser theorems? Clearly, they do not hold in general since some Church-Rosser systems can have "unbounded delay." Also, one could have bounded delay but some sort of looping behavior that could give rise to no tight bound existing for timing. Nevertheless, it would be interesting to characterize, for general Church-Rosser systems, when various types of bounds on timing hold.

Acknowledgement

The authors wish to thank A. Meyer, M. Paterson, and L. Valiant for pointing out that an earlier version of Theorem 5.1 could be strengthened.

References

[1] A. Church and J. B. Rosser, "Some properties of conversion," Trans. Am. Math. Soc. 39 (1936) 472-482.

[2] S. N. Cole, "Real-time computation by iterative arrays of finite-state machines," Doctoral Thesis, Report BL-36, Harvard University, 1964.

[3] H. B. Curry and R. Feys, Combinatory Logic, North-Holland Publ. Co., Amsterdam, 1958.

[4] P. C. Fischer, "Generation of primes by a one-dimensional real-time iterative array," JACM, 12, No. 3, (1965) 388-394.

[5] R. W. Floyd in D. E. Knuth, The Art of Computer Programming, 3, p. 241, Problem 36.

[6] R. M. Keller, "A fundamental theorem of asynchronous parallel computation," Third Annual Sagamore Conf. on Parallel Computation, 1974.

[7] E. F. Moore, "The firing squad synchronization problem," Sequential Machines, Selected Papers, Addison-Wesley, (1964) 213-214.

[8] B. K. Rosen, "Tree-manipulating systems and Church-Rosser theorems," JACM, 20, No. 1 (1973) 160-187.

[9] G. Rozenberg and A. Salomaa, Editors., Lecture Notes in Computer Science, 15, L Systems, Springer-Verlag, 1974.

[10] A. R. Smith, "Real-time language recognition by one-dimensional cellular automata," JCSS, 6, (1972) 233-253.